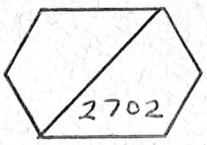
From Proc. of Infotech State of the Art Conf., Copenhagen, '77.

Burstall



DESIGN CONSIDERATIONS FOR A FUNCTIONAL PROGRAMMING LANGUAGE

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
45 BANBURY ROAD
OXFORD

INTRODUCTION

As hardware gets cheaper, people become increasingly aware of the high cost of software and particularly of those components of the cost which are due to bugs in programs and to inflexible programs. So it really would be delightful to have a simple programming language in which one had a good chance of avoiding mistakes and in which programs were transparent enough to be altered easily. I believe this goal is worth working towards even if it means radical changes in our programming habits. But simplicity does not come easily; it needs many years of hard thought and much work on refining concepts.

In this paper I will first list some aspects of programming languages which contribute to accuracy and flexibility. I will then illustrate them by describing a functional language, called NPL, which I have been developing at Edinburgh with the help of some of my colleagues. It is a vehicle for experimentation towards the above goals rather than a finished language. However it will serve to make my points more concrete. Such a language pays penalties of inefficiency just because it is remote from the machine level. So it might better be thought of as a specification language or as a language for making a first try at a programming project. We are doing some work on ways of making it more efficient.

An important aspect of programming not yet dealt with in NPL is data abstraction. I will indicate why this is necessary for a satisfactory system.

ASPECTS OF PROGRAMMING LANGUAGE DESIGN CONTRIBUTING TO CORRECTNESS AND FLEXIBILITY

The following points seem important in a language for writing correct and flexible programs. They are listed rather briefly but most of them are illustrated in the description of NPL which follows.

No assignment: - referential transparency

Functional languages which work in terms of expressions and their values, using recursion instead of loops, seem much clearer and less error prone. The expressions are transparent in the sense that their value depends only on their textual context and not on some notion of computational history. Each variable is given a value just once where it is produced. Assignments which alter data structures are particularly prone to cause bugs; disallowing them greatly simplifies the language although it does slow down execution.

Storage control using a garbage collector

It is a great simplification if storage allocation is entirely automatic and storage is retrieved automatically using a garbage collector. Using stack storage rather than a heap prevents procedures producing data structures as results and this removes much of the power of the procedure mechanism.

Maximum use of user defined types

The user should define his own types wherever possible, thus type 'date' or type 'age' rather than type 'integer'. The machine should check these types. The language should allow type variables so that code can be as general as possible, for example 'list of alphas' rather than specifically 'list of numbers', where alpha is a type variable which can be instantiated to any type.

Exhaustive case analysis

Each data type will have a set of disjoint subtypes, each made with a different constructor function, e g lists are made with *cons* or *nil*. It should be easy to do case analysis with respect to these constructors and easy for the compiler to check that the analysis is exhaustive. This avoids mistakes like forgetting to put in the test for the empty list.

Avoiding unnecessary names

It should not be necessary to introduce a large number of different names for functions to operate on data structures, to test what subtype it is (e g is it made by *cons* or *nil*) or select components (e g *car* or *cdr*). We can use just the constructor name and by using pattern matching syntax let it denote testing or selecting as the context dictates.

Abolishing explicit recursion as far as possible

Given that we want an assignment-free language, that is, a functional style of programming, we will use recursion instead of loops. However, undisciplined use of recursion is not at all desirable. It is much better to use standard constructions related to the data structures, for example, set expressions of the form $\{f(x) : x \text{ in } S \text{ and } p(x)\}$ instead of recursion over the set S. In a language which allows higher order functions, that is, procedures taking procedures as parameters, and which allows lambda expressions to denote functions, we can easily define useful higher order functions to 'package' recursion and save writing them explicitly; for example we may define a function map, which takes as arguments a function from As to Bs and a list of As and which produces as result a list of Bs, thus map $(\lambda x.x^2, \{1,2,3,4\}) = \{1,4,9,16\}$. (This higher order facility is not available in NPL but we plan to provide it.)

Making sequences into data objects

In traditional imperative-style programming, sequences are represented as successive values of a variable, thus a sequence of integers is represented as values of an integer

variable. It is also possible to represent them by an array or list, but this has the disadvantage that all the elements of the sequence have to be in the store at the same time. An idea aptly called 'lazy evaluation' by Henderson and Morris (011) (due originally to Wadsworth (015)) enables us to deal with such sequences as lists but have them evaluated one element at a time as needed. Amongst other things, this enables one to achieve the same effect as co-routines in an imperative language but in a more perspicacious way, by passing a lazily evaluated list as the result of a function. It also enables one to do conversations or processes in a functional language.

Modularity and data abstraction

The main trick in getting large programs free from bugs is to write them as separate pieces, each as self-contained as possible, with the pieces communicating with each other in an explicit manner. An important technique for this is data abstraction (see for example Hoare in Chapter 2 of (009)). This has been explored as an aid to good programming style in such recent languages as ALPHARD (016,014) and CLU (013), which may be thought of as developing one aspect of the SIMULA class concept (008). An abstract data type, such as set or queue, is defined in terms of:

a The primitive procedures which create its elements and operate on them, and
b The axioms which these procedures must obey.

For the abstract type these procedures are declared but their bodies (definitions) are not given, only the axioms about them. The code which uses the procedures relies solely on these axioms and has no access to the representation of the data elements. This representation is specified elsewhere in the bodies of the procedures.

Conventional block structure scope rules make it difficult to maintain the desired separation of use and representation, so these languages adopt different rules for identifier scopes.

Recently J A Goguen and I have been considering a rather more general technique for program structuring in terms of 'theories', each referring to several data types and the operations over them. We point out in (007) that not only programs, but also program specifications, need to be structured into self-contained pieces.

VERIFICATION AND TRANSFORMATION

An important aim of language design is to make it easier to verify that a program meets a given specification. In this respect functional languages seem to offer considerable advantages, and powerful verification systems have been written by Boyer and Moore (002,003) and by Aubin (001). Another aid to the development of correct programs is the technique of program transformation, whereby a simple but inefficient starting program is transformed into an acceptably efficient one by steps which maintain its correctness. Again functional programs lend themselves very well to transformations (006); indeed our work on transformations was the major stimulus for the development of the NPL language, which is now used in our transformation system.

AN EXPERIMENT IN PROGRAMMING LANGUAGE DESIGN: NPL

NPL is a programming language, or rather an incomplete fragment of a programming language, which I designed and implemented as a private project. It grew out of the work on program transformation which John Darlington and I had been doing. We wanted a simple language in which programs were easy to transform, and we decided upon the recursion equation form for our system (006). We also wanted a language for which it would be easy to do correctness proofs. I extended somewhat the notation we had been using and wrote a parser, type-checker and interpreter for the language, NPL, in POP-2. These were modified and extended over a period of time and now form a rather ungainly implementation, but one which has enabled us to try out the language. Darlington connected up his implementation of our transformation system and this can now transform NPL programs into other, hopefully more efficient, ones.

Jerry Schwarz wrote an interpreter allowing 'lazy evaluation' (011) and this can be used instead of the normal interpreter.

The aim throughout was to design a programming language which was very simple and encouraged clarity and manipulability of programs. It uses well known ideas from a variety of sources.

The language bears a close resemblance to the OBJ language of Goguen (010). I believe it is incomplete and inadequate because it lacks data abstraction facilities. Goguen and I have been working together on ways to provide such facilities; the first fruit of our labours is a proposal for a specification language CLEAR (007). We hope to develop a programming language to go with it.

NPL also, regrettably, lacks facilities for defining higher-order functions, although I do not believe it would be too hard to put them in. It needs some other conveniences, notably sensible output facilities.

The language is still an experimental vehicle and no claim is made that the definition or implementation is completely bug-free.

Major influences in the design of NPL have been LISP and Landin's ISWIM (012). I have also been influenced by experience with POP-2 (005). Burge (004) gives a good account of programming in a functional style.

Kinds of statement

The language has five kinds of statement, each introduced by a key word or sign.

1	Data structure declarations	data
2	Function type declarations	dec
3	Variable type declarations	var
4	Function definitions	---
5	Precedence declarations	pre or inf (prefix or infix)

Declaring data structures

All data is represented as 'terms' built up from 'constructor functions', for example *cons*(*a*,*cons*(*b*,*nil*)) in which *cons* is a binary constructor and *a*, *b*, *nil* are nullary

ones (that is, constants). These constructor functions are uninterpreted, they just construct.

Functions are constrained to take inputs and outputs of specified 'type'. These types are created by the programmer and the data structure declaration introduces a new type together with the constructor functions which create elements of that type. For example:

data *num* <= *zero* ++ *succ*(*num*)

says that *num* is a new data type (natural number) and it has two constructors, *zero*, a nullary, and *succ*, a unary, taking a natural number as argument. So the elements of *num* are *zero*, *succ*(*zero*),*succ*(*succ*(*zero*)),..., that is, 0,1,2,... . The sign ++ indicates disjunction.

To introduce a type 'list-of-numbers' we say

data *nlist* <= *nnil* ++ *ncons*(*num*,*nlist*)

Thus an *nlist* is either *nnil*, or it is formed by the constructor *ncons* acting on a *num* and an *nlist*.

Now we need more general kinds of list than just lists of numbers. So we allow variables denoting types. They are declared thus:

var *alpha*, *beta*, *gamma* : type

We can use these variables in data statements thus:

data *list*(*alpha*) <= *nil* ++ *cons*(*alpha*, *list*(*alpha*))

Here *list* is not a type but a one-argument type constructor. We can now use *nil* and *cons* for lists of any type and no longer need special functions for lists of numbers.

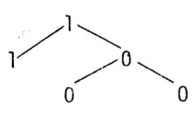
Another example is a binary tree with labels on the nodes, thus

data *bintree*(*alpha*) <= *tip*(*alpha*)++ *node*(*bintree*(*alpha*), *alpha*, *bintree*(*alpha*))

The term

node(*tip*(*succ*(*zero*)), *succ*(*zero*),
node(*tip*(*zero*), *zero*, *tip*(*zero*)))

is a type *bintree*(*num*) and may be pictured as



Of course there are certain data types provided when you enter the system so that every user does not have to re-define them for himself. There is even a little syntactic

sugar to make terms easier to write, although this could not have been defined in the language itself. The predefined types include *num*, *list* and *truval* defined by

data *truval* <= *true* ++ *false*.

The numbers 0,1,2,3, are available with their usual meanings. Any function symbol can be made a prefix or an infix with a given precedence, thus

inf 3 +, - pre 4 not

For lists, the infix :: is provided instead of the clumsier *cons* used above, so we may write 0::(1 :: (2 :: *nil*)). As a further convenience one may use brackets for constructing lists, thus

[0, 1, 2] or [2+2, 3+3]

Other predefined types are *sets*, *association lists* and *n-tuples*. Sometimes we want to declare two or more mutually recursive data types, say *s* and *t*. We do it in this style:

data *s*<= *a* ++ *c*(*t*); *t* <= *b* ++ *d*(*s*)

where *a*, *b*, *c*, *d* are the constructors.

Declaring the type of functions

Having declared some types, we want to declare some functions to operate on them. We do this using dec thus

dec *append* (*list*(*alpha*), *list*(*alpha*)) <= *list*(*alpha*)

or assuming we have already made + an infix as above

dec *num* + *num* <= *num*

A function symbol may be declared several times with different argument types.

(As identifiers for functions, variables or types we may use either an alphabetic character, possibly followed by some alphanumerics, or else a sequence of signs from the set

+, -, *, /, =, <, >, :, @, $;

for example, dog99 or <*>.)

Defining functions

Having declared the type of a function, we must define it. A function is defined by one or more clauses, each starting with the symbol ---. The clauses use variables which must have been previously defined by a variable declaration. For example,

```
var m, n: num
--- 0 + n <= n
--- succ(m) + n <= succ(m + n)
```

This defines addition, for example

```
succ(succ(0)) + succ(succ(0))
= succ(succ(0) + succ(succ(0))) = succ(succ(0 + succ(succ (0))))
= succ(succ(succ(succ(0)))),
```

so 2 + 2 = 4.

The left hand side is the function symbol being defined with patterns as arguments, where a pattern is made from (distinct) variables and constructor functions. The right hand side can be any expression using the variable introduced on the left but no others (except for local variables in where clauses, as explained below). A variable, once declared, stays declared and keeps its type, and it may be used in many function definitions (compare the usage in mathematics).

The pattern in the defining clauses should exhaust the possibilities given in the data statement introducing the type, thus any number must be of the form 0 or the form *succ*(*m*). To define append:

```
var a, a1 : alpha                          var a1, a11 : list(alpha)
--- append(nil, a11)  <=  a11
--- append(a :: a1, a11) <= a :: append(a1, a11)
```

To define reverse so that *rev*((1, 2, 3)) is (3,2,1)

```
dec rev(list(alpha)) = list(alpha)
--- rev(nil) = nil
--- rev(a :: a1) = append(rev(a1),  [a])
```

We may qualify a right hand side by a condition, thus

```
dec member(alpha, list(alpha)) <= truval
--- member(a1, nil)             <= false
--- member(a1, a :: a1)         <= true if a1 = a
                                <= member(a1,a1)  if not

(the general form is --- ... <= ... if ...
                     <= ... if ...
                      ...
                     <= ... (if not)
```

We may introduce local variables using where, thus

```
dec f(num) <= num
--- f(n)   <= g(i, i) where i == n + n
```

(equivalent to *g*(*n* + *n*, *n* + *n*) but the addition is only done once). A pattern may be used after where, thus if *c* is a binary constructor function,

```
i + j + j  where c(i,j) == f(k)
```

evaluates *f*(*k*) and expects to get a term of the form *c*(*t1*, *t2*,); *i* + *j* + *j* is then evaluated with *i* and *j* bound to *t1* and *t2* respectively.

If a function symbol has been declared several times, defining clauses are given for each (some time after the declaration) and the type checker will determine which definition is to be used, using the types of the argument expressions. Thus we could use + for appending lists as well as adding numbers.

Set expressions

It is quite easy to introduce sets into the language just as we did lists

data *set(alpha)* <= *nilset* ++ *consset(alpha, set(alpha))*

In this way sets are in fact isomorphic to lists but distinguished from them by type checking. We can then define functions appropriate to sets such as membership, union, intersection and inclusion.

To make it easier to work with sets, we permit a notation very similar to mathematical usage (following Schwarz in his SETL language). Suppose for example that we already have constants *s* and *t* denoting sets of numbers, then a typical set expression would be

{*i* + *j* : *i* in *s*, *j* in *union(s,t)* & *i*<*j*}

The general form is:

{*expression* : *variable* in *expression* & *expression*, ...,
variable in *expression* & *expression*}

We also allow explicit sets like {1,2,3}. So

{*i*+*j* : *i* in {1,2,3}, *j* in {1,2,3} & *i* = *j*} is {2,4,6}

(Remark. Because of character set problems on our system we have to use '<:' for '{' and ':>' for '}' at the moment).

We can also write quantified expressions such as

all *i* in {1,2,3} : *i*>0

which evaluates to true, since *i*>0 holds for each *i* in {1,2,3}. The general form is

all *variable* in *expression*: *expression*

where the first expression denotes a set and the second a truth value. We may replace all by exists.

Program

A program consists of a sequence of declarations and definitions enclosed within def ... end followed by an expression to be evaluated enclosed within val ... end, or any sequence of such elements. When val ... end is encountered, the machine evaluates the expression and prints the result.

An example program is given in the next section.

Lazy evaluation and infinite data structures

A version of NPL exists using lazy evaluation; it was written by my colleague Jerald Schwarz.

Lazy evaluation gives in functional languages the benefits that co-routines give in imperative ones: a procedure does not have to produce its result all at once, but can pass it on a little at a time.

The standard method of applying a function to its arguments is 'call by value': first you evaluate each argument, then you evaluate the body of the function using these argument values. For lazy evaluation you just evaluate as much of the argument expressions as you need during computation of the function body; once you have done some evaluation you remember the partial result (unlike 'call-by-name'). It turns out that this can be regarded as extending the language to deal with infinite data structures, such as the list [0, 1,2, ...]; this does not sound useful until you realize that the input from a teletype can well be thought of as a (potentially) infinite list.

Consider for example a function *successors*(*m*,*n*) which produces the list [*m*,*m*+1, ..., *m* + *n* - 1] and a function *squares* which squares each element in a list so that *squares* ([x_1, ..., x_n]) = [x_1^2, ..., x_n^2]. Suppose we evaluate *squares*(*successors*(1,4)). Now *successors* (1,4) is [1,2,3,4] and *squares*(*successors*(1,4)) is [1,4,9,16], but we do not need to evaluate all of *successors*(1,4) to get to the first element of the answer 1^2, only the first element 1. Consider the function which produces (potentially) the infinite list of all successors.

```
dec allsuccessors(num) <= list(num)
--- allsuccessors(m)   <= m :: allsuccessors(m + 1)
```

Now if we evaluate *squares*(*allsuccessors*(1)) we really need a lazy evaluator, because it is very unwise to try to calculate the whole list *allsuccessors*(1) before computing and printing the square of its first element.

Compare the calculation in school arithmetic of 5718 divided by 2; 2 into 5 goes 2 and 1 left over, 2 into 17 goes 8 .... We develop the successor digits one at a time, getting better and better approximations to the answer. This is really necessary for .5 divided by 3 which has the infinite answer .16666 ... . Just as we use finite sequences of decimal numbers to approximate to real numbers, we can use finite initial parts of lists to approximate to finite or infinite lists, thus [1, and [1,2 and [1,2,3 are better and better approximations to the infinite list [1, 2, 3, 4 ... . In the above example done in lazy evaluation mode *squares* asks for better and better approximations to its argument and produces better and better approximations to its result. In practice the approximations are represented by a chain of list cells with a bit of code hung on the end; this code will compute further elements of the list when they are needed.

So with lazy evaluation we can write a computation in separate phases, even if the first phase seems to produce a large intermediate result, since this will not be there all at once. Perhaps more important is that lazy evaluation enables a functional language to deal with real-time exchange of information. For a conversational program we can let the teletype input be the (infinite) argument list and have a part of the resulting (infinite) output list typed as each input is read and dealt with. Thus functional languages can conveniently handle processes.

Lazy evaluation is slower than 'call-by-value' evaluation by a constant factor which may be between two and ten (we have not compared implementations precisely).

PROGRAMMING IN NPL: AN EXAMPLE

The following example shows how we can use NPL to implement two data types *set* and *partition*. We give the data declarations and a set of primitive functions for each of these, using a particular representation (not a very efficient one). In point of fact sets are already provided by the system in a library file, together with the syntactic sugar of {....} expressions. We might well need other functions on sets and partitions not defined here, but the ones given will illustrate the technique.

```
DEF

VAR ALPHA: TYPE          VAR A, A1: ALPHA

PRE 10 SET
DATA SET ALPHA  <= NILSET ++ CONSSET(ALPHA, SET ALPHA)
VAR AS,AS1:SET ALPHA

INF 6 :::
DEC ALPHA ::: SET ALPHA          <= SET ALPHA
DEC MEMBER(ALPHA, SET ALPHA)     <= TRUVAL
DEC SET ALPHA + SET ALPHA        <= SET ALPHA
DEC UNIQUE(SET ALPHA)            <= ALPHA

--- A:::AS                       <= AS IF MEMBER(A,AS)
                                 <= CONSSET(A,AS) IFNOT
--- MEMBER(A1,NILSET)            <= FALSE
--- MEMBER(A1,CONSSET(A,AS))     <= TRUE IF A1=A
                                 <= MEMBER(A1,AS) IFNOT
--- NILSET+AS1                   <= AS1
--- CONSSET(A,AS)+AS1            <= A:::(AS+AS1)
--- UNIQUE(CONSSET(A,NILSET))    <= A

END


DEF

DATA PARTITION <= PAR(SET SET NUM)

DEC MERGE(NUM,NUM,PARTITION)     <= PARTITION
DEC UNITPART(SET NUM)            <= PARTITION
DEC NATMAP(NUM, PARTITION)       <= SET NUM
DEC EQUIV(NUM,NUM,PARTITION)     <= TRUVAL

VAR L,M,N:NUM  VAR LS,MS,NS: SET NUM  VAR LSS,MSS,NSS: SET SET NUM
VAR P:PARTITION

--- UNITPART(LS)          <= PAR((: (:  L :) : L IN LS:))
--- MERGE(L,M,PAR(LSS))   <= PAR((LS+MS):::NSS)
                             WHERE NSS == (: NS: NS IN LSS
                                    & NOT(MEMBER(L,NS)) AND
                                      NOT(MEMBER(M,NS)) :)
                             WHERE LS == NATMAP(L,PAR(LSS))
                             WHERE MS == NATMAP(M,PAR(LSS))
--- NATMAP(L,PAR(LSS))    <= UNIQUE((: LS : LS IN LSS & MEMBER(L,LS) :) )
--- EQUIV(L,M,P)          <= NATMAP(L,P) = NATMAP(M,P)

END
```

Figure 1: An NPL program

The NPL program is in Figure 1; it is explained below.

Sets

First we declare the data structure *set*, with constructors *nilset* and *consset*, using a type variable *alpha*; *nilset* is the empty set and *consset* adds an element to a set. We declare variables for alphas and for sets of alphas. Next we declare four primitive functions on sets; ::: and + are infixes but + has already been given a precedence when we introduced it for arithmetic. These functions are

:::	*a*:::*as* means {*a*} ∪ *as*; it acts like *consset* but checks that *a* is not already a member of *as* before adding it.
member	set membership
+	set union
unique	selects the unique member of a singleton set; unique ({*a*})=*a*

Note that unlike *consset*, ::: is not a constructor, and cannot be used on the left-hand side of a defining clause. For general use we need a function split which inverts ::: by taking a non-empty set and giving an element and the remainder.

Partitions

We use sets to define a new data type *partition*. A partition of a set is a set of subsets of it which are disjoint from each other and together exhaust the set, for example {{1,2},{3,4}} is a partition of {1,2,3,4}.

So a partition is not just *any* set of sets, but one satisfying some constraints; a function taking a partition as argument should not accept just any set of sets. To make the distinction we define a one argument constructor, *par*, which makes a partition from a set of sets. Of course we can use *par* on the left of a definition clause to get the set of sets back; it is just a marker, a little protective box. Now we can define a small collection of functions which can get at what is in the box and do all the necessary operations for partitions. Other functions on partitions which we define thereafter can just use these partition 'primitive' functions and need not refer to the set of sets or use *par*. Indeed we should be able to enforce this and prevent them accessing the inside of *par*. This is the essence of data abstraction in a language, but is not currently provided in NPL. (It would be rather easy to provide it; for example by having the word abstraction after data in a data declaration would indicate that the constructor or constructors in the declaration were to be local to the def ... end in which the declaration occurs and not available outside it). Meanwhile we can just obey the restriction as a matter of self discipline; then so long as we check that the partition primitives create only proper partitions (disjoint and exhaustive) our whole program will be sure to do so.

The primitives are:

unitpart	makes a partition of singleton sets, e g *unitpart*({1,2,3}) = *par*({{1},{2},{3}})
merge	makes a new less refined partition from a given one by merging two sets e g *merge*(1,2,*par*({{1},{2},{3}})) = *par*({{1,2},{3}}) (these two functions are the only ones which create partitions)

natmap the 'natural mapping' which tells us which set in a partition contains a given element
e g *natmap*(1,*par*({{1,2},{3}})) = {1,2}

equiv tells whether two elements are in the same set in the partition
e g *equiv*(1,2,*par*({{1,2},{3}})) = true

Remark on data abstraction

The facilities provided in NPL for definition of parameterized data types, such as set alpha, have one very important omission. The parameter alpha may be instantiated by any data type but the primitives for set alpha need an equality operator over alphas (member uses it). In the code above, the general = operator is used; this is correct for numbers but if alpha is itself a set we should use the set *equality operator*, which ignores the ordering of the elements. Currently NPL has no way of associating such an operator with a type; to provide one would involve a rather major revision, but it is certainly necessary to do so.

REFERENCES

001 AUBIN R
Strategies for mechanizing structural induction
Proc Fifth Intl Joint Conf on Artificial Intelligence Cambridge Mass pp 363-369 (1977)

002 BOYER R S and MOORE J S
Proving theorems about LISP functions
JACM vol 22 no 1 pp 129-144 (1975)

003 BOYER R S and MOORE J S
A lemma driven automatic theorem prover for recursive function theory
Proc Fifth Intl Joint Conf on Artificial Intelligence Cambridge Mass pp 511-519 (1977)

004 BURGE W H
Recursive programming techniques
Addison-Wesley Reading Mass (1975)

005 BURSTALL R M, COLLINS J S and POPPLESTONE R J
Programming in POP-2
Edinburgh University Press (1971)

006 BURSTALL R M and DARLINGTON J
A transformation system for developing recursive programs
JACM vol 24 no 1 pp 44-67 (1977)

007 BURSTALL R M and GOGUEN J A
Putting theories together to make specifications
Proc Fifth Intl Joint Conf on Artificial Intelligence Cambridge Mass pp 1-45-1058 (Invited paper) (1977)

008 DAHL O-J, MYHRHAUG B and NYGAARD K
The SIMULA 67 common base language
Publication S22 Norwegian Computing Centre, Oslo (1970)

009 DAHL O-J, DIJKSTRA E W and HOARE C A R
Structured programming
Academic Press (1972)

010 GOGUEN J A and TARDO J
OBJ-0 preliminary users manual
Semantics and Theory of Computation Report UCLA Los Angeles (1977)

011 HENDERSON P and MORRIS J
A lazy evaluator
Third Symp on Principles of Programming Languages Atlanta pp 95-103 (1976)

012 LANDIN P J
The next 700 programming languages
CACM vol 9 pp 157-166 (1966)

013 LISKOV B, SNYDER A, ATKINSON R and SCHAFFERT C
Abstraction mechanisms in CLU
CACM vol 20 no 8 pp 564-576 (1977)

014 SHAW M and WULF W A
Abstraction and verification in Alphard: defining and specifying iteration and generators
CACM vol 20 no 8 pp 553-564 (1977)

015 WADSWORTH C P
Semantics and pragmatics of the lambda calculus
PhD thesis Programming Research Unit Oxford (1971)

016 WULF W A, LONDON R L and SHAW M
An introduction to the construction and verification of Alphard programs
IEEE Trans on Software Engineering vol SE-2 no 4 pp 253-265 (1976)